Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory
Communication Systems Group

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Search in Peer-to-Peer Networks

Semester Thesis

presented by

Daniel Reichle

ETH Zurich, Switzerland

Supervisor:

Dipl.-Ing. Jan Mischke,
Dipl.-Ing ETH David Hausheer,
Prof. Dr. Burkhard Stiller

of the

Computer Engineering and Networks Laboratory

July 2003

Abstract

The topic of this semester thesis is an implementation of SHARK (Symmetric Redundant Hierarchy Adaption for Routing of Keywords). The SHARK algorithm was developed by Jan Mischke and Burkhard Stiller and is described in [1].

SHARK is an algorithm that defines the build-up of a peer-to-peer (P2P) network through successive insertion of nodes that store objects. The essential point is, that these objects are described through meta-data. Arranging the nodes according to the meta-data of their objects enables an efficient search for objects in the SHARK network.

The implementation of the SHARK system is realized in four layers to make in particular the communication on the UDP layer exchangeable. The topmost layer is the user interface (UI), which is implemented as a command line input interpreter. The SHARK algorithm is implemented on the second layer, where the node is the central component. A node stores SHARK objects and accesses the routing table. For communicating with other nodes, it hands over SHARK messages to the underlying communication layer. In this layer, the structure of the SHARK messages get flattened for propagation to the socket layer. At the receiver side, the incoming UDP packets are handed over to a dispatcher that decides what is to be done with the respective packet. To do so, the appropriate SHARK message is reassembled from the received one-dimensional data. If the received packet poses a task to the node, a handler is instantiated that is responsible for whose processing.

With the implemented main functionality of SHARK, it is possible to create a network through successive insertion of nodes, to insert objects and to find objects through queries. The program has been tested with small, manageable networks and the working of the implemented functions has been shown. By this means, the feasibility of the SHARK algorithm has been proven.

Kurzfassung

Der Inhalt dieser durch den voliegenden Bericht beschriebenen Semesterarbeit ist eine Implementierung von SHARK (Symmetric Redundant Hierarchy Adaption for Routing of Keywords). Der SHARK-Algorithmus wurde von Jan Mischke und Burkhard Stiller entwickelt und ist in [1] beschrieben.

Zweck von SHARK ist der Aufbau eines Peer-to-Peer (P2P) Netzwerks, in welchem ein effizientes Suchen nach gespeicherten Objekten möglich ist. Dabei werden Objekte mittels Metadaten in mehreren Dimensionen und Leveln beschrieben. Jedes zu unterscheidende Merkmal repräsentiert in den Metadaten eine Dimension (falls es sich bei den Objekten um Musikstücke handelt, kämen als mögliche Dimensionen beispielsweise das Erscheinungsdatum oder die Stilrichtung in Frage). Die Unterscheidung in den einzelnen Dimensionen wird ausserdem mit jedem Level verfeinert. Dabei kann die Anzahl der Dimensionen von Level zu Level variieren. Jeder Knoten im Netzwerk soll nun einer Metadatenbeschreibung zugeordnet werden und Links zu Objekten mit den gleichen Metadaten speichern, so dass die Objekte dann über ihre Metadaten effizient gefunden werden können. Damit dies möglich ist, werden die Knoten im Netzwerk ihren Metadaten entsprechend angeordnet.

Das Routen erfolgt sequenziell den Metadaten entsprechend. Begonnen wird auf dem ersten Level in der ersten Dimension. Die Anfrage wird dann an einen Knoten mit dem entsprechenden Eintrag auf dieser Position seiner Metadaten weitergeleitet und dort mit den Einträgen der Routing Tabelle auf dem ersten Level in der zweiten Dimension verglichen usw. Das Ziel ist, dass jeder Knoten für jede Metadatenposition für jeden möglichen Wert einen Nachbarn kennt, an den er Anfragen weiterleiten kann. Hierfür erhält jeder neu hinzukommende Knoten für jede Metadatenposition die entsprechenden Einträge der Routing Tabelle des jeweils kontaktierten Knotens, der ja bereits Bestandteil des Netzes ist. Auf diese Weise wird die für das Routen nötige Information über das ganze Netzwerk mit ausreichender Redundanz verteilt, so dass jeder Knoten Anfragen auf der obersten Stufe entgegennehmen und weiterleiten kann.

Eine Anfrage nach einem Objekt wird zuerst bis zu einer sogenannten Group of Interest (GoI) geleitet. Eine GoI ist ein oder mehrere Knoten, die Links auf Objekte mit den gleichen Metadaten unterhalten. Hat eine Anfrage einen Endknoten erreicht, wird von ihm aus die ganze GoI geflutet. Für das Auffinden eines gesuchten Objektes ist auf dieser Stufe ein String erforderlich, der auf dem letzten Level der Metadaten nicht mehr eine Kategorie des Objektes beschreibt, sondern das Objekt eindeutig identifiziert.

Die Implementation des SHARK Systems wurde in vier Schichten realisiert, um insbesondere die Kommunikation auf UDP-Ebene austauschbar zu machen. Der SHARK Algorithmus ist in der zweitobersten Schicht implementiert, wobei der Knoten die zentrale Komponente darstellt. In ihm werden die SHARK Objekte gespeichert, und durch ihn erfolgen die Zugriffe auf die Routing Tabelle. Für die Kommunikation mit anderen Knoten übergibt er SHARK Nachrichten an die nächst tiefere Schicht. Dort wird die Struktur der SHARK Nachrichten für die Weitergabe an die Socket-Schicht geglättet. Auf der Empfängerseite werden die ankommenden UDP Pakete einem Dispatcher übergeben, welcher entscheidet, was mit dem jeweiligen Paket geschieht. Zu diesem Zweck wird aus den empfangenen eindimensionalen Daten wieder die entsprechende SHARK Nachricht erstellt. Stellt die empfangene Nachricht einen Auftrag für den Knoten dar, wird ein Handler instanziert, der für dessen Ausführung verantwortlich ist.

Die oberste Schicht schliesslich stellt das User-Interface dar, welches in dieser Implementation als Befehlszeilen-Eingabe realisiert ist.

Implementiert ist die Grundfunktionalität von SHARK, die es ermöglicht ein Netzwerk durch sukzessives Einfügen von Knoten aufzubauen, Objekte einzufügen, sowie Objekte durch Anfragen zu finden. Getestet wurde das Programm nur mit kleinen, überschaubaren Netzwerken. Zudem dienten die Tests ausschliesslich der Verifizierung der implementierten Funktionen und nicht als Beweis der Lauffähigkeit unter allen möglichen Bedingungen.

Die Implementierung beweist, dass die Grundfunktionalität des SHARK Algorithmus durch ein Programm praktisch realisiert werden kann.

Contents

1 Introduction **1**

2 Implementation of SHARK **2**

2.1 Related Work and Problem Solving Approach 2
2.2 Overview of the SHARK Algorithm 2
2.3 Overview of the Software Architecture 4
2.4 User Interface 5
2.5 SHARK System Layer 5
2.5.1 SHARKNode class 5
2.5.2 SHARKMetadata class 9
2.5.3 SHARKObject class 10
2.5.4 RoutingTable/RTableEntry class 10
2.6 Message Passing Layer 10
2.6.1 SHARKMessage classes 10
2.6.2 SHARKMessageCommunicator class 12
2.6.3 MsgList 12
2.6.4 Dispatcher class 12
2.6.5 IDcreator class 13
2.6.6 Hash class 13
2.6.7 MsgTypeResolver class 13
2.7 Network Connection Layer 13
2.7.1 Sender class 13
2.7.2 Receiver class 13
2.8 Global Classes 13
2.8.1 Constants class 13
2.9 Exceptions 14
2.10 Test class 14

3 The Command Line Interface: A User‘s Guide **15**

3.1 System prerequisites 15
3.2 Installation 15
3.3 Usage 15

4 Conclusions and Future Work **19**

5 Appendix **20**

5.1 Deviations of the Implementation from the SHARK Specification 20

6 References **21**

7 List of Figures **23**

8 List of Tables **25**

1 Introduction

Peer-to-Peer (P2P) networks are getting more and more popular. They increasingly replace client/server applications in areas like P2P computing, collaboration, trading, network testing and - maybe on the largest scale - filesharing. The main advantage of P2P networks compared to client/server systems is that the tasks of the central server is distributed to many computers. That way usage of idle PC resources makes expensive servers unneeded, which furthermore are critical components in the network. But through the absence of a central component, there are also disadvantages arising. Particularly distributed lookup and search of objects is highly complex and not yet satisfactorily solved [2].

Several concepts have been proposed to improve search in P2P systems, one of which is SHARK (Symmetric Redundant Hierarchy Adaption for Routing of Keywords). While other concepts like AGILE [3] provide a search mechanism through routing based on a hash value of the requested object, SHARK applies the approach of routing towards objects directly based on keywords. To make this keyword routing possible, SHARK - which is primarily targeted at file-sharing applications - works with objects that are described through meta-data. Because the meta-data determines where the according object belongs to, it is possible to map the meta-data structure to the network structure and the meta-data can be used as an address for the stored objects. This means that for each node in the network, a certain meta-data description is assigned, and a node stores links to objects that are described through the same meta-data. Due to the lack of a central infrastructure, the routing information has to be redundant, so that every node can take over the role of the starting node for a query. To achieve this, a new node, that wants to join the network, copies the according routing table row of each hop on the way it gets routed to its place in the network at its insertion.

The goal of this semester thesis was to implement SHARK in Java, namely to construct a generic SHARK meta-data hierarchy, to implement the basic mechanisms to construct the search hierarchy, to create the routing tables, to insert nodes and objects and to route toward objects based on their meta-data description.

This paper presents the resulting implementation in Section 2, the usage of the appropriate command line interface in Section 3 and a conclusion as well as a discussion of open issues in Section 4.

2 Implementation of SHARK

2.1 Related Work and Problem Solving Approach

To get an idea of how an implementation of SHARK would look like, I first ensured my understanding of how the algorithm works. The next step was to divide the problem into smaller subproblems. During the study of the algorithm, it soon became clear, that a major part of the implementation will care about sending, receiving and processing SHARK messages. The most obvious was to adopt the idea of a common network stack and thus to divide the software into several layers to make the actual algorithm as much independent from the underlying network as possible. Another advantage of this approach was to allow a collaboration with another student who was working on an implementation of AGILE [4], namely to use the same low level communication modules in both implementations. Unfortunately, unexpected changes in the requirements lead to individual changes in the code, which made an exchange of the modules unsuitable. This was not too bad- because I looked at the communication modules as the precondition for the other parts of the software, this was the first part that was implemented. So the re-writing of this part allowed me to make changes to its interface, so that it suits better to the requirements of the meanwhile designed higher layer. Nevertheless, some parts of this early collaboration are still contained in the code, what I will refer to in the code description to explain inconsistent looking code.

As denoted above, while the design of the software was created in a top-down manner, the implementation was made bottom-up. This made successive testing of the implemented functionality possible. Except for the communication modules, changes in the design were not necessary during the implementation. For simplicity reasons, at some points the implementation slightly differs from the description of the algorithm, but without violating the functional principle of SHARK.

The result is a working program which implements the basic functionality of SHARK. The details of the implementation will be described in the remainder of this section.

2.2 Overview of the SHARK Algorithm

The goal of SHARK is the build-up of a P2P network, which allows an efficient search for objects stored therein.The approach chosen in SHARK is to describe objects through meta-data in several dimensions and levels. Each attribute that we want to differ is represented by a dimension (if the objects are songs for example, possible dimensions would be the date of release and the genre). The categorization in the different dimensions is improved by each level. Furthermore, the number of dimensions can vary from level to level. Each node in the network is assigned to a specific meta-data description and stores links to objects described by the same meta-data, so that objects can be found efficient through their meta-data. To make this possible, the nodes are arranged in the network according to their meta-data.

Routing is done sequential corresponding to the meta-data. It gets started at the first level in the first dimension. The request is then forwarded to a node representing the according entry on that position in its meta-data. At that node, the meta-data of the request gets compared at the second dimension to the node‘s according routing table entries, and so on. The goal is that each node knows for each meta-data position for each possible value a neighbor node to which it can forward requests. Therefore, each node that wants to join the network gets the according routing table entries for each meta-data position from the respective contacted node, who - of course - is already part of the network. By this means, the information needed for the routing gets distributed over the whole network with sufficient redundancy, and each node is able to accept and forward initial requests on the topmost level.

A request for an object is routed at first to a so called Group of Interest (GoI). A GoI consists of one or more nodes that maintain links to objects with the same meta-data. If a request reaches an end node, it gets forwarded to the whole GoI through flooding. To find the object at this level, a string is needed that does not describe any category of the object, but that identifies the object definitely.

2.3 Overview of the Software Architecture

The implementation is divided into 4 layers as Figure 1 illustrates. The topmost layer implements the user interface, the SHARK system layer implements the SHARK algorithm, the message passing layer acts as an intermediate between the SHARK system and the network connection layer, which contacts any underlying network.

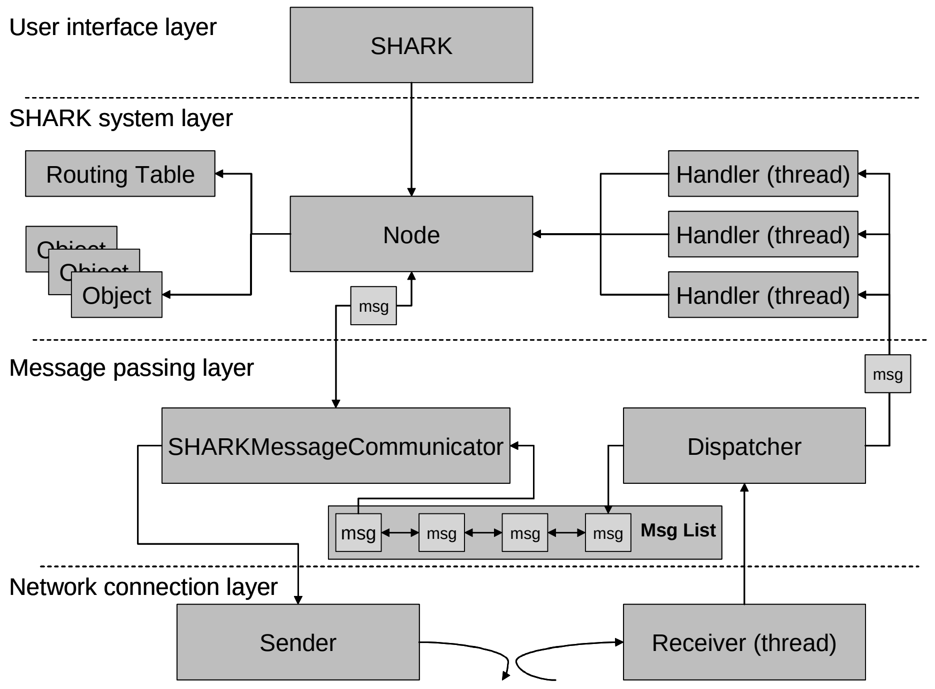

Figure 1: Software architecture overview

The user interface consists of only one class called ‚SHARK‘ which defines the entry point of the application. From here, a SHARK node is instantiated and operations like its insertion into a SHARK network, creation and insertion of objects and queries for objects are invoked.

The SHARK system layer contains the ‚SHARKNode‘ class as the central component as well as the appropriate routing table. The ‚SHARKNode‘ class implements all methods necessary to manage a SHARK network, primarily for inserting itself into the network, for inserting objects and for querying. SHARKObjects are referenced by the node and belong also to this layer. The different handler-threads are instantiated by the dispatcher and are therefore structurally not really part of the SHARK system layer. But because they invoke methods in the node while processing tasks from incoming messages, they functionally belong also to this layer. SHARK messages, used for communication with other nodes, are implemented as classes containing all needed information in member fields.

The task of the message passing layer is to forward SHARK messages sent by the node to the network connection layer, and to process network packets received by the network connection layer. While the network connection layer just implements a socket interface for sending and receiving UDP packets and does not care about their meaning to the SHARK system, the message passing layer needs knowledge about the semantics of their payloads. The sending process is rather simple: the node sends a SHARK message to the SHARKMessageCommunicator that flattens the data contained in the message, and passes it to the sender. Flattening means that all values in the member fields of the SHARK message are extracted and filled in a one-dimensional array that is needed by the sender to build a UDP packet. When a packet arrives, its payload is handed over by the receiver to the dispatcher. The dispatcher looks at the received array as a flattened SHARK message. It reads the type field in the message header, builds the respective SHARK message, and decides what has to be done with it. Dependent on its type, it adds the SHARK message to the message list, where the SHARKMessageCommunicator is looking for received messages, or instantiates a handler that carries out a task transmitted by the message. The reason why the handlers are needed is that the dispatcher has to be ready to process the next incoming packet while a handler can work on the task as a thread.

The network connection layer consists of a sender and a receiver. The sender is instantiated every time a packet has to be sent. It builds a UDP socket, fills the data received by the SHARKMessageCommunicator in a UDP packet and sends the packet to the requested network address. The receiver is instantiated only once when the node is created. It also controls a UDP socket by which it catches incoming UDP packets in an endless loop. If a packet arrives, it hands the payload over to the dispatcher and waits for the next packet.

In the following, I will explain the structure and functionality of the used classes in detail in the same top-down order as in the outline above.

2.4 User Interface

The user interface, implemented in the class ‚SHARK‘, is just a small interface to invoke the functions of the SHARK system in an interactive way. It is realized as a command line interpreter. The only method, the ‚main‘ method, executes an endless loop that displays an input prompt and parses the string entered by the user for commands to build a SHARK node, insert the node into a SHARK network, build a SHARK object, insert the object into the network or search for an object stored in the SHARK network.

The provided user interface is not very convenient and serves only for testing the program. For a user‘s guide of the interface, see Section 3.

2.5 SHARK System Layer

The classes in this layer, namely ‚SHARKNode‘, ‚SHARKMetadata‘, ‚SHARKObject‘, ‚RoutingTable‘ and ‚RTableEntry‘ are all derived from logical units of the algorithm. As I mentioned before, the handler classes belong from a structural point of view to the message passing layer, and are therefore discussed later in a separate section.

2.5.1 SHARKNode class

Node creation

To instantiate a new SHARKNode, its constructor has to be called with a receiver port and the node‘s meta-data as parameters. The third parameter is a boolean, that sets the ‚showTraffic‘ variable in the ‚Constants‘ class, to turn the output concerning the message traffic on or off. A few instructions for initializing to node are then executed: First of all, the node stores the given meta-data as its own meta-data. It then looks for his own IP address (the local address of the computer where the program runs), takes the port number given by the caller, and builds a routing table entry containing this information. He then creates a new routing table of a default initial size (see the description of the RoutingTable class in Section 2.5.4 for details) and fills in his own entry for every position of its own meta-data.

After initializing the list for storing SHARK objects and two buffers for storing message IDs, it creates a SHARKMessageCommunicator and a Receiver. After starting the receiver, the node is ready to communicate with other nodes and informs the user via standard output that it was successfully initialized.

Node insertion

To insert the new node into an existing SHARK network, one has to know the network address of a node that is already part of the network. With the IP address and the UDP port number of this known host, the ‚insert_me‘ method can be invoked. This is not necessary for the first node in the network, but for each subsequent node. The message sequence chart implemented in this method is shown in Figure 2.

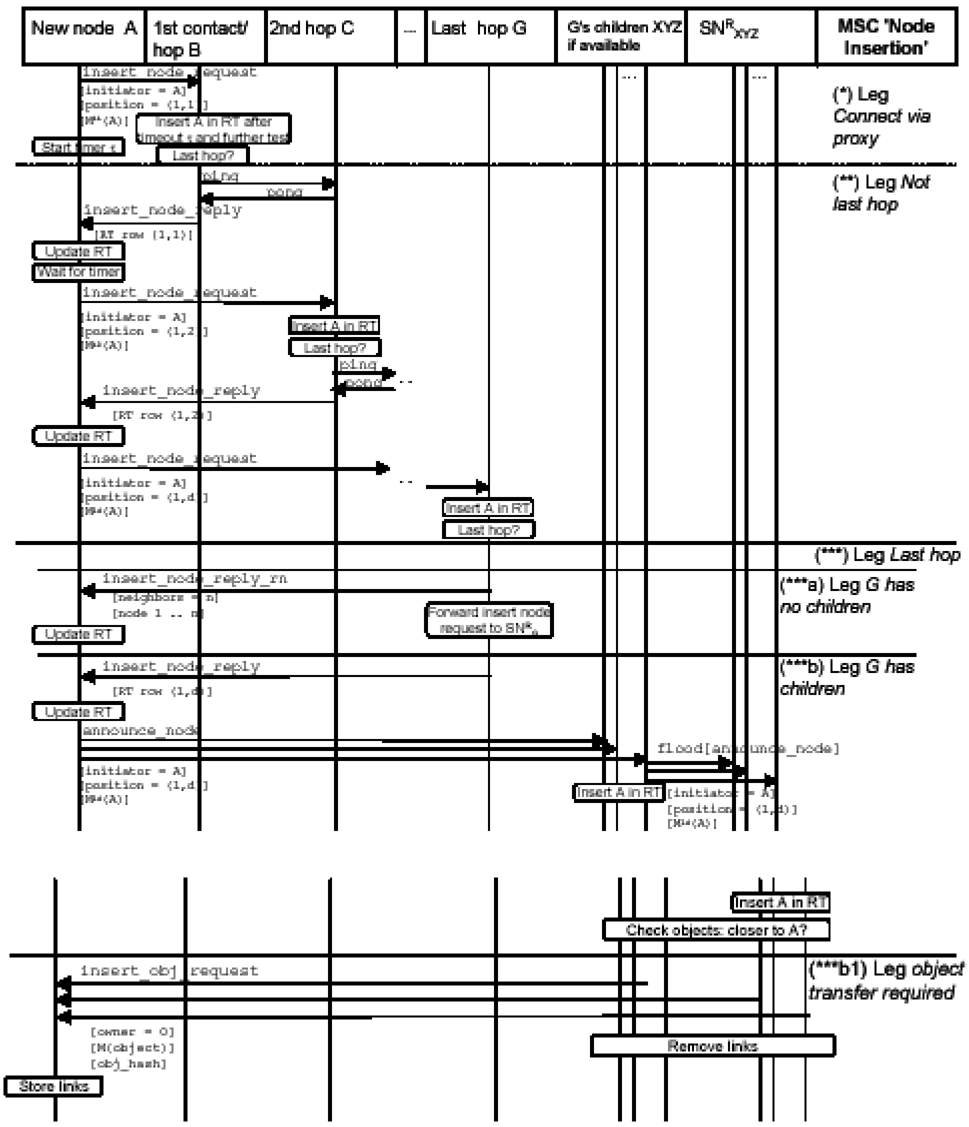

Figure 2: SHARK MSC Node Insertion [5]

Our node sends the first insert_node_request to the indicated host. In this message, the node sends its meta-data value of position (1,1), which means the value stored in its meta-data on the first level in the first dimension. It also indicates the position, so that the contacted node knows, which row of its own routing table it has to return to our node. In the node, that receives the message, the ‚handle_insert_node_request‘ method is invoked. It adds our node to its routing table and returns its routing table row of the position requested by our node. In this routing table row, returned in a insert_node_reply message, our node tries to find a node with the same meta-data on that position. If it does find such a node, this node is the next peer to be contacted. Our node adds the received routing table entries to its own routing table and sends the next insert_node_request to the next hop, now for position (1,2). This continues until either no matching entry is found in the returned routing table row, which means that our node is the first of a new GoI, or we did arrive at the end of our node‘s meta-data structure. In the first case, our node has to inform the last contacted hop about the new GoI. It does this through sending an announce_message to that hop and to all nodes in the received routing table row. The announce_message is then flooded by its receivers to their respective GoIs, so that they all get knowledge about our node and are able to route subsequent requests to our GoI. If some of these nodes store objects that belong to the new GoI, objects are transferred to the new node. In the latter case, we have found a node that belongs to the GoI that we were looking for. This node also knows that our node belongs to its GoI and therefore it will send a insert_node_reply_rn message, which informs our node about its GoI neighbors. However, the ‚insert_me‘ method is not responsible for this case. Instead, an insert_node_reply_rn message invokes a handler when it is received. This handler then carries out the task of adding our new neighbors. This case does not suit exactly to the specification, where the sending of an insert_node_reply before the insert_node_reply_rn message is not designated. However, this does not change the algorithm- it is just to simplify the implementation, because our node does not have to check for each reply, whether it is a common reply, or a reply_rn message. The ping/pong messages are not implemented. This mechanism is to improve performance by not sending links to nodes that are not alive.

Object insertion

For inserting an object, the ‚insert_object‘ method is called. It requires a SHARK object and a replication factor. This factor indicates, how many nodes are to store links to this object. The ‚insert_object‘ method just builds an insert_object_request message and sends it to its own network address to start the routing of the object. This causes the ‚handle_insert_object_request‘ method to be called. The routing mechanism described in the following is illustrated by Figure 3.

In the ‚handle_insert_object_request‘ method, the node first checks, if the object is to be inserted by itself. If so, it stores the object in its objects list and sends a replicate_link message to the appropriate number of GoI neighbors, if the replicate factor indicated in the object insert request message is greater than zero. If the meta-data of the new object does not suit to the node‘s meta-data, it looks up all suitable next hops to forward the message to. If there is no suitable entry in the routing table, the GoI for to the new object does not yet exist. In this case, the node stores the object in its own objects list and sends a replicate_link message to its GoI neighbors. Such objects will be transferred to the appropriate GoI at the time the first node belonging to that GoI is inserted. The decision that objects have to be transferred is made in the ‚handle_announce_node‘ method. To transfer an object, the ‚insert_object‘ method can be called with a specific network address and the replication factor set to zero. That way, the object is directly inserted at the appropriate node without routing the insert_object_request to the same node again.

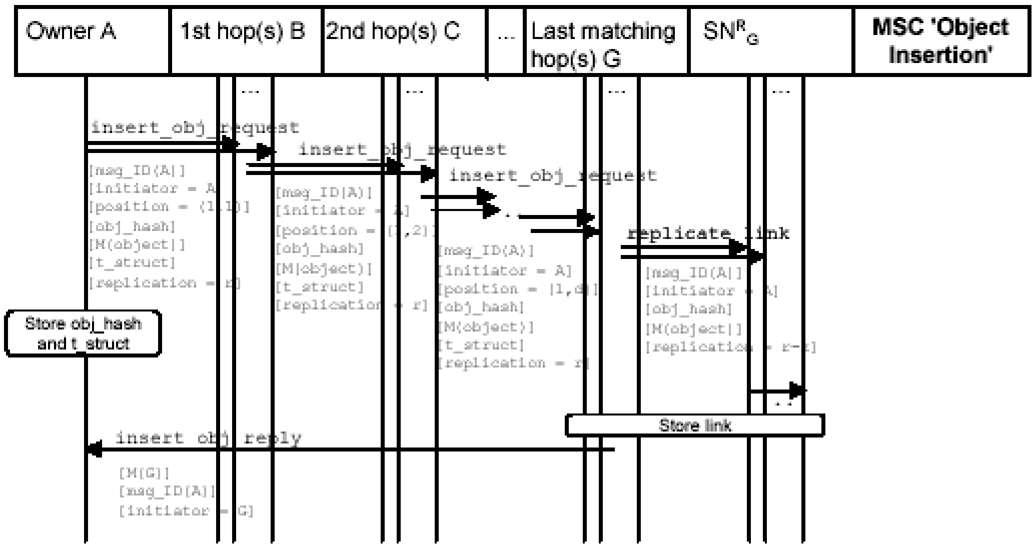

Figure 3: SHARK MSC Object Insertion [5]

Query

If the user wants to search an object, he calls the ‚query‘ method. Similar to the ‚insert_object‘ method, this method is just the starting point of a query, while the subsequent process is done through successive calls of the ‚handle_query_request‘ method in the hops on the way the query is routed through the SHARK network. The ‚query‘ method takes the meta-data description of the requested object as its input parameter, builds a query message and sends it to its own network address, what causes a call of the ‚handle_insert_object_request‘ method. The query processing mechanism described in the following is illustrated by the message sequence chart in Figure 4.

The query routing is handled by the ‚handle_insert_object_request‘ method, as mentioned before. In that method, several different cases have to be examined, and it is therefore rather complex. First of all is checked, whether the same query has been received earlier. A query is identified by its message ID and its request position (level, dimension). If the ‚recent_queries‘ buffer does not contain this query, the query is added to the buffer, and its processing is started. The node reads the requested meta-data on the requested position. If this value is zero, which stands for ‚unspecified‘, it forwards the query to all nodes it knows on the next level in the hierarchy as well as to his GoI neighbors, and checks its objects list for objects matching the query. If there are any, it sends them in a query answer message to the initiator of the query. When looking for matches, the node only compares the meta-data without the string - this part is still to be implemented.

The next step is to check if the position of the query request is the last position of the meta-data hierarchy. If so, the node checks its objects list for objects matching the query, sends them in a query answer message to the initiator, forwards the query to all GoI neighbors, and stops the processing of this query.

If the position of the query request is not the last position of the meta-data hierarchy, the node sends a new query to all nodes matching the next position of the meta-data hierarchy. If there is no matching entry in the node‘s routing table, the query is for an object of a non-existent GoI. Also in this case, our node has to look at its objects, send matching objects in a query answer to the initiator of the query, and forward the query to its GoI neighbors.

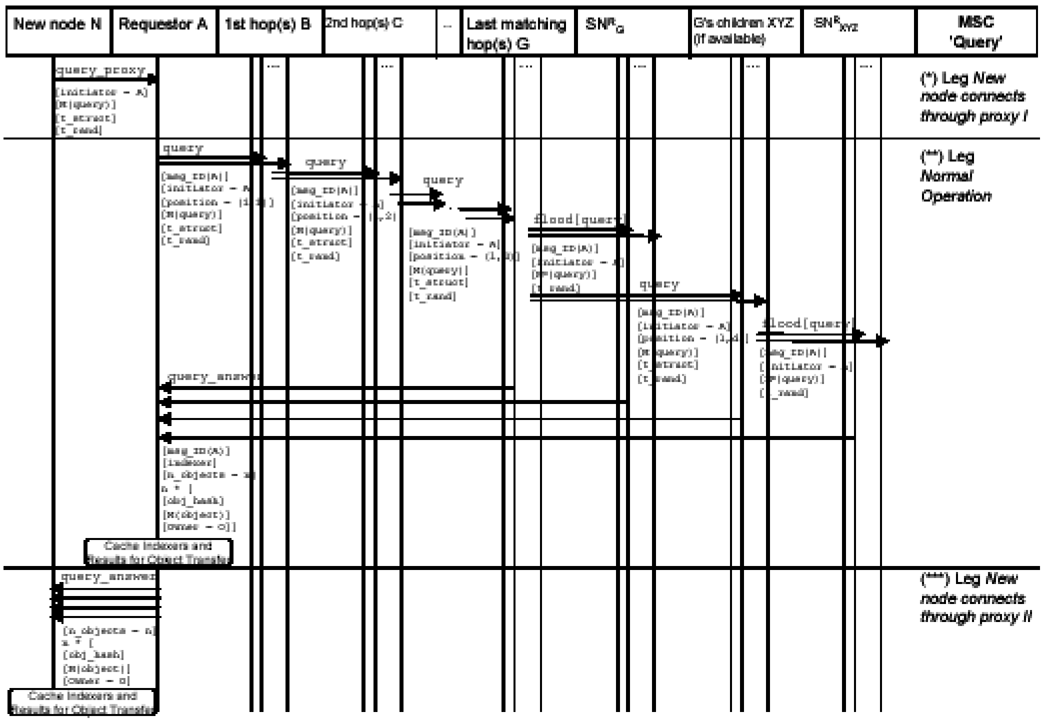
New node N
Requestor A
1st hop(s) B
2nd hop(s) C
Last matching hop(s) G
SNR_G
G's children XYZ (if available)
SNR_XYZ
MSC 'Query'
(*) Leg New node connects through proxy I
(**) Leg Normal Operation
(***) Leg New node connects through proxy II
query_proxy
query
flood[query]
query_answer
Cache Indexers and Results for Object Transfer

Figure 4: SHARK MSC Query [5]

When the node receives a query answer message, the ‚handle_query_answer‘ method is invoked. How such answers are to be handled is not implemented- the handler just informs the user via standard output about which node has found which object.

2.5.2 SHARKMetadata class

The meta-data class stores the meta-data for an object in a two-dimensional array called ‚levels‘. This data structure represents the meta-data as introduced by the description of the SHARK algorithm. The first dimension of the array are the different ‚levels‘ of the meta-data, the second dimension are the different ‚dimensions‘, whose number can vary from level to level.

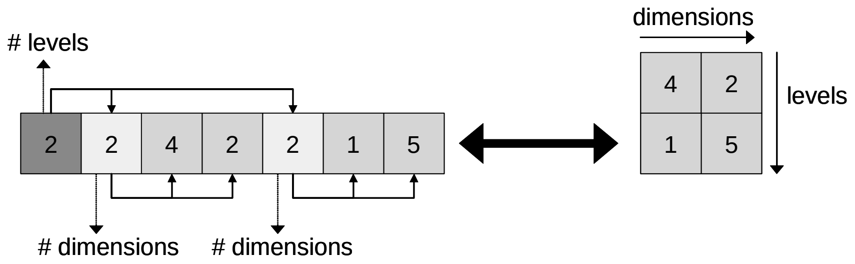

Figure 5: 1- and 2-dimensional representation of the meta-data

In the implementation, there are two possible representations of the meta-data, as illustrated in Figure 5. One is the two-dimensional described above. The ‚SHARKMetadata‘ class uses this representation, because access to the data is simpler that way. For other purposes, namely for sending a meta-data description in a SHARK message over the network, this representation is not suitable, so we also need a one-dimensional representation. The one-dimensional representation of the meta-data does not suit exactly to the specification of SHARK. While in the specification, each value is preceded by its position, in the implementation, the respective position of the values is implicitly indicated. The second difference to the specification is, that while in the specification, a meta-data description also contains a string that definitely describes an object, in the implementation, this string is not part of the meta-data structure. When passing meta-data, the string is handled separately.

For instantiating the ‚SHARKMetadata‘ class, the data has to be passed in its one-dimensional representation. To get the data back in the one-dimensional representation, the class provides methods to get the needed array size and to get the data as a one-dimensional array.

2.5.3 SHARKObject class

SHARK objects that are stored in the SHARK network are represented through the ‚SHARKObject‘ class. A SHARK object is described through its meta-data description, a string (see preceding paragraph), a hash value, and the network address of its owner. The meta-data and the string are assigned to the object at its instantiation. The hash value is calculated from the string, and the network address of its owner is assigned at the insertion of the object. The node that inserts an object into the network is the owner of that object.

The method for calculating the hash is implemented, but not yet as provided for in the specification. To clearly identify an object, the hash value is to be built of the binary data of the corresponding file. But the use of files as objects was beyond the scope of this implementation, so the hash value of an object is built of the object‘s name.

2.5.4 RoutingTable/RTableEntry class

The ‚RoutingTable‘ class controls all accesses to a node‘s routing table. The table is implemented as a 3-dimensional array which is initialized with an initial size specified by a constant. The first dimension corresponds to the levels, the second dimension corresponds to the dimensions of the respective meta-data description. In the third dimension, routing table entries of type ‚RTableEntry‘ are stored. These entries are links to the node‘s neighbors on the respective position (level/dimension). For each occurring meta-data value, at most two entries are stored. Any further entry for the same value will replace the oldest of the existent. After initialization, in each level, the number of dimensions corresponds to the number of levels, and all entries are filled with zeros. This is for performance reasons, because a newly created table would have to be extended for every new entry, if the initial size was zero. That way, extension is only necessary when the table size exceeds the initial size, which would not so often be the case, if the appropriate initial size is chosen.

Each table entry contains a meta-data value and the appropriate network address.

2.6 Message Passing Layer

2.6.1 SHARKMessage classes

SHARK messages are used for the communication between SHARK nodes. The message format used is illustrated in Figure 6. There exist different types of SHARK messages that differ in the structure of the payload (Message Body).

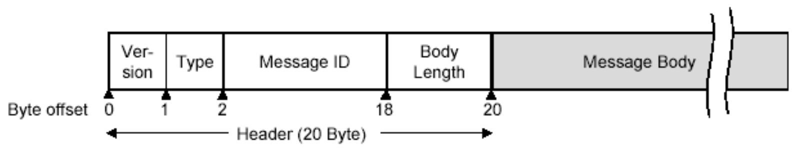

Figure 6: SHARK Message Format [5]

The general message format is implemented as a class named ‚SHARKMessage‘, and all classes for messages of other types inherit from that class.

The structure of these classes is always the same: they all have two constructors, one is called with class structures as parameters, the other is called with an array of bytes as the only parameter. The first constructor is used for creating of outgoing messages by a higher level method. The received parameters are then filled in the payload array. The second constructor is used for incoming messages by the dispatcher.

Moreover, the message classes provide methods that read the requested information out of the payload array and return the respective information as class structures or primitive types according to the type of the requested information.

The payload structure of the used SHARK messages are illustrated in Figure 7.

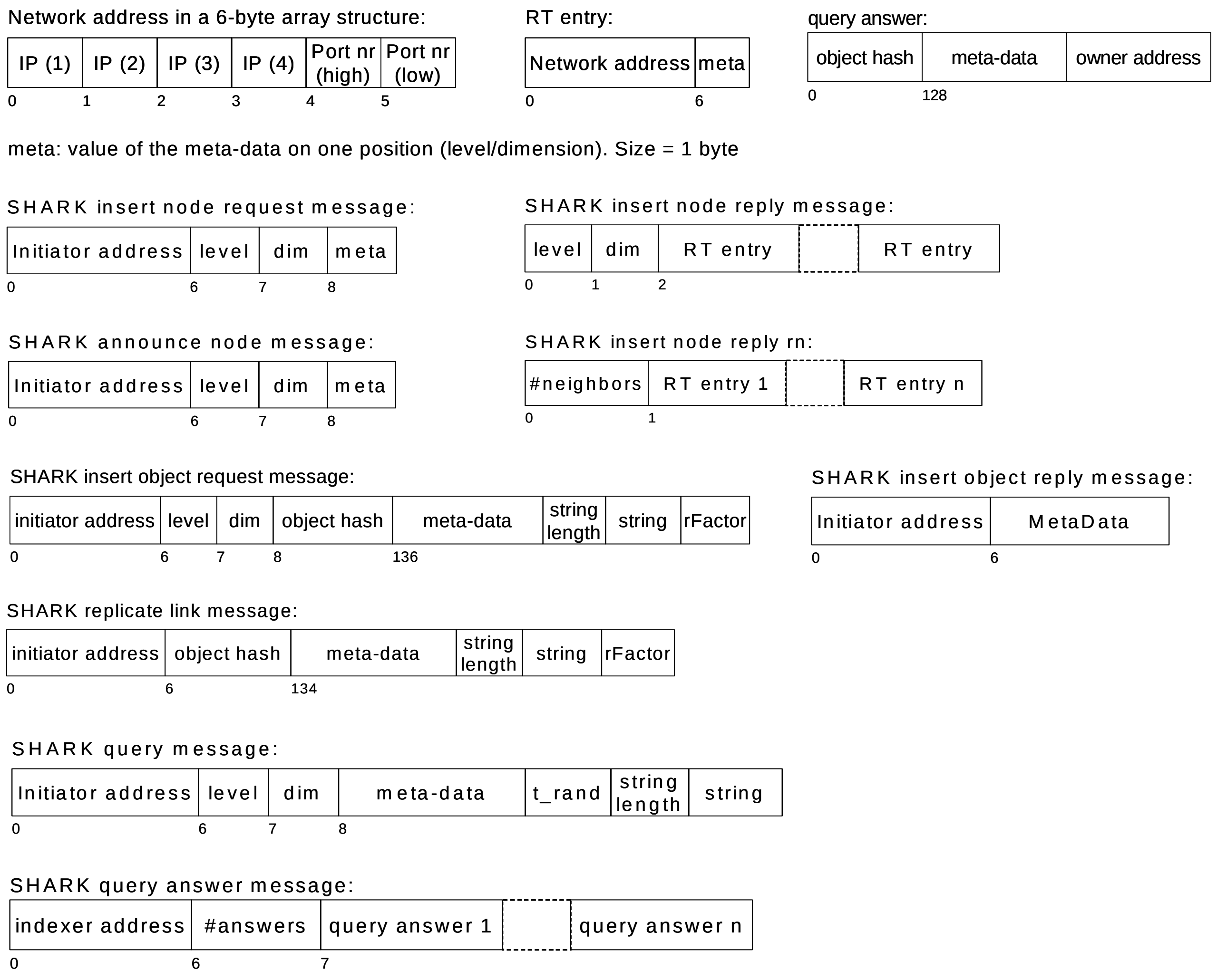
Network address in a 6-byte array structure:
IP (1) | IP (2) | IP (3) | IP (4) | Port nr (high) | Port nr (low)
0 1 2 3 4 5
RT entry:
Network address | meta
0 6
query answer:
object hash | meta-data | owner address
0 128
meta: value of the meta-data on one position (level/dimension). Size = 1 byte
SHARK insert node request message:
Initiator address | level | dim | meta
0 6 7 8
SHARK insert node reply message:
level | dim | RT entry | | RT entry
0 1 2
SHARK announce node message:
Initiator address | level | dim | meta
0 6 7 8
SHARK insert node reply rn:
#neighbors | RT entry 1 | | RT entry n
0 1
SHARK insert object request message:
initiator address | level | dim | object hash | meta-data | string length | string | rFactor
0 6 7 8 136
SHARK insert object reply message:
Initiator address | MetaData
0 6
SHARK replicate link message:
initiator address | object hash | meta-data | string length | string | rFactor
0 6 134
SHARK query message:
Initiator address | level | dim | meta-data | t_rand | string length | string
0 6 7 8
SHARK query answer message:
indexer address | #answers | query answer 1 | | query answer n
0 6 7

Figure 7: SHARK message payload formats

One SHARK message type is missing in the figure above: the acknowledgment message. This message has a payload of length zero.

2.6.2 SHARKMessageCommunicator class

The ‚SHARKMessageCommunicator‘ class provides the two methods ‚send_msg‘ and ‚receive_msg‘.

The ‚send_msg‘ method takes a network address and a SHARK message as input parameters. It extracts the payload array from the SHARK message and hands it over to the sender, together with the address of the receiver. If the sent message is not an acknowledgment message, the communicator waits for the appropriate acknowledgment message. It does so by asking the message list - where incoming acknoledgement messages are stored - every 500 milliseconds, if an acknowledgment message with the same message ID as the sent message has arrived. If the appropriate acknowledgment message has not arrived until a time-out (defined as constant in the ‚Constants‘ class) is reached, he throws an exception. Otherwise, the sending process was successful.

The ‚receive_msg‘ method takes an IP address and a byte, that indicates the requested SHARK message type, as input parameters. It asks the message list every 500 milliseconds, if a SHARK message of the requested type has arrived from the requested IP address. If the requested message is found before the time-out (also defined as constant in the ‚Constants‘ class) is reached, it is returned to the caller and removed from the message list. Otherwise, an exception is thrown.

2.6.3 MsgList

The ‚MsgList‘ class controls all accesses to the message list, where incoming passive messages are stored. ‚Passive messages‘ means messages, that are expected by the node, instead of posing a task to it (ex. acknowledgment messages). Its basic functionality is adding messages to the list, searching messages in the list, and removing messages from the list.

2.6.4 Dispatcher class

The ‚Dispatcher‘ class implements only one method accessible from outside the class: the ‚dispatch‘ method. And this is in fact all that this class does: dispatching messages handed over by the receiver. As inputs, the dispatch method takes the payload of a UDP packet and the IP address and return port of the sender. The dispatcher sees the received byte array as a SHARK message, reads the message type and ID fields from the header of the message and creates the appropriate SHARK message class. Dependent on the message type, it then adds the SHARK message to the message list (e.g. insert_node_reply type), or creates a handler thread, to which it hands over the SHARK message (ex. insert_node_request type). To the handler thread, it also hands over a reference to the ‚SHARKNode‘ class that instantiated the dispatcher. This enables the handler thread to access methods in the ‚SHARKNode‘ class (callback mechanism). The only thing a handler does, is invoking the corresponding ‚handle_xxx‘ method in the ‚SHARKNode‘ class with the SHARK message as the parameter. After starting the handler thread, the dispatcher does not have to care about the message any longer, and is ready to process the next incoming packet.

If the received message is not an acknowledgment message, the dispatcher causes the ‚SHARKNode‘ class to send an acknowledgment message to the sender of the received packet.

2.6.5 IDcreator class

This little class provides the method ‚create_id‘ that takes a byte array as an input parameter and fills it with random bytes. This method is used, for creating new message IDs that clearly identifies a SHARK message. The ID is used to identify duplicate messages for example at flooding the GoI.

2.6.6 Hash class

This class is taken over from the implementation of AGILE (see [4]). It takes a string and an integer value as input and calculates the MD5 hash value of the string. The length of the resulting hash value is determined by the integer value.

2.6.7 MsgTypeResolver class

This class is to get the respective string for any message type. The class is used by the dispatcher and the message communicator that display the message traffic via standard output for testing purposes.

2.7 Network Connection Layer

2.7.1 Sender class

The ‚Sender‘ class contains two constructors and the ‚send‘ method. The first constructor takes an integer as input which it binds to a new UDP socket. The second constructor is called without any parameters and binds a default port number (defined in the ‚Constants‘ class) to the socket. By default, the send port number is just the receive port number increased by 1.

The ‚send‘ method is called with an IP address, a port number, and a byte array containing the data to be sent. The sender creates a UDP packet and sends the data to the requested network address.

2.7.2 Receiver class

The ‚Receiver‘ class has to be instantiated with a port number and a reference to a ‚SHARKNode‘ class as parameters. It creates a dispatcher, to which it hands over the reference to the node for the callback mechanism described above.

The receiver is a thread, and therefore implements a method called ‚run‘. This method is called when the thread is started. Here, a UDP socket is created, before the receiver enters an endless loop. In this loop, the receiver just listens on its port. If a packet arrives, the receiver hands whose payload over to the dispatcher, together with whose sender‘s network address, and waits for the next packet.

2.8 Global Classes

2.8.1 Constants class

As its name indicates, this class stores all constants used in the program. All constants stored herein are declared static, so they can be accessed from any class.

Some of the constants are derived from the SHARK specification and should not be changed. Others are parameters like time-outs or the initial size of the routing table and are set arbitrarily. Which constants may be changed, and which ones should better not be changed is indicated by the comments in the source code.

A member of the ‚Constants‘ class (that does not really belong here) is the boolean ‚showTraffic‘, that is set when the node was built with parameter showTraffic=true. This boolean is considered by methods that produce output that shows the message traffic. This output is not produced, if this boolean is set to ‚false‘.

2.9 Exceptions

The following user defined exceptions are used:

- MyException: This exception was planned to be a multi-purpose exception. It is therefore still used where a specific exception is not yet implemented. The idea was, to have only one type of exception, with a string that describes what went wrong. When caught, the string is sent to the standard output. This is quite nice for developing, but not very flexible. Other types of exception has then been created.
- ReceiveAckTimeoutException: Thrown by the ‚send_msg‘ method in the ‚SHARKMessageCommunicator‘ class when the time-out exceeds while waiting for the acknowledgment message.
- ReceiveMsgTimeoutException: Thrown by the ‚receive_msg‘ method in the ‚SHARK-MessageCommunicator‘ class when the time-out exceeds while waiting for the requested SHARK message.
- NoEntryFoundException: Thrown by the ‚getNextHop‘ and ‚getAllNextHops‘ methods in the ‚RoutingTable‘ class, if no entry is found in the requested position of the routing table, that matches the requested meta-data.
- PositionOutOfRangeException: Thrown by the ‚getNextHop‘ and ‚getAllNextHops‘ methods in the ‚RoutingTable‘ class, if the caller tries to look for an entry in the routing table at a position that does not exist.

2.10 Test class

The class ‚Test‘ contains a routine that builds up a SHARK network of 9 nodes, inserts some objects and starts a query to search an object. This routine has been used for testing the functionality of the code. For testing purposes and as an example for the usage of the system, the ‚Test‘ class is still part of the code and is invoked by simply entering ‚test‘ at the command line prompt of the user interface described in the following section. Note, that the routine can only be executed once. Because the lack of a mechanism implemented to disconnect the receiver socket, the only way to disconnect the socket is to terminate the program. Therefore, at the end of the ‚test‘ routine, a ‚System.exit(0)‘ call is executed, and the program has to be restarted for the next use.

3 The Command Line Interface: A User‘s Guide

3.1 System prerequisites

To run the SHARK system, Java 2 Runtime Environment has to be installed on the computer. The program has been tested with version 1.4.1, so its working with lower versions can not be guaranteed.

3.2 Installation

An installation is not necessary. To run the program, just do the following steps:

1 Start the command line interpreter (cmd.exe) of your windows system and change the directory to the directory, where you have stored the ‚SHARK.jar‘ file.

2 Enter the following command:

```
>java -cp SHARK.jar SHARK
```

Note that names in the Java environment are case sensitive, so you have to type ‚SHARK‘ in uppercase letters. The application indicates that it is running by displaying the command prompt:

```
$:
```

3.3 Usage

The SHARK system now expects you to create a node. You have two possibilities to create a node:

Node creation with extended output function

To build a node, enter ‚node‘ followed by a port number and ‚-m‘ as a second parameter:

```
$: node <port> -m
```

The second parameter is used to turn on the extended output functions that shows you all the incoming and outgoing message traffic of your node.

Node creation without extended output function

If you don‘t like to analyze the message traffic, just leave out the second parameter and type the command like this:

```
$: node <port>
```

That way you can test the SHARK system without any flickering output messages getting on your nerves.

Note that the application manages only one node. If you like to run more than one nodes on the same computer, you have to start a new command line interpreter for each node and repeat the steps described above, using a different port number.

Entering meta-data

After entering the ‚node‘ command, the system prompts you to enter some meta-data. The meta-data has to be entered as a list (for the description of the format see Section 2.5.2.).

Here is an example for entering a meta-data description with 2 levels and 2 dimension in each level:

```
$: node 10020

Enter metadata: 2,2,1,3,2,4,7
```

Note, that the values have to be separated by commas.

The node then prints his network address, meta-data and routing table to tell you that its initialization is completed:

```
*******************************************************************
New node built. IP address is /192.168.0.1 receiving at port nr 10020
Metadata: (2 levels)
  Level 1: 1 3
  Level 2: 4 7

Routing table of /192.168.0.1 port 10020:
  [1] 1 192.168.0.1:10020 [2] 3 192.168.0.1:10020
  [1] 4 192.168.0.1:10020 [2] 7 192.168.0.1:10020
*******************************************************************

$:
```

The routing table has to be read as follows: Each row on the output represents a level. The dimensions are indicated by their numbers enclosed in square brackets, so ‚[1]‘ indicates that the following entries belong to the first dimension. An entry is a meta-data value, followed by an network address (IP address : port number). So, the entry

```
[2] 7 192.168.0.1:10020
```

means, that in the 2nd dimension of the to level corresponding to the row (in this case the 2nd level), our node has a neighbor with IP address 192.168.0.1 that receives packets at port number 10020 and has a meta-data value of 7 on that position (level 2, dimension 2). Here, of course, this is our node itself- it is the only node in our network up to now.

You can get this data to your screen any time you like, by typing ‚info‘:

```
$: info
```

Inserting a node into a SHARK network

With just one node, no network can be built- so we have to repeat all the steps described above on another computer, or at least on another command line interpreter on the same computer.

Note: If you want to run more than one node on the same computer, you have to use different receive ports for the different nodes. Note also that a node‘s send port number is always the number of the receive port increased by 1. So do not just increase the receive port numbers by one when building more than one node on the same computer.

The first node does not have to be inserted. For all following nodes type

```
$: insert <IP address> <port>
```

The entered IP address and port number can be of an arbitrary SHARK node that was built in advance. In my example, I have built another node and insert the former at it:

```
$: insert 192.168.0.1 10022
node inserted successfully! (new GoI)
```

Inserting an object

The syntax for inserting an object is similar to the creation of a node. First, type ‚obj‘,

```
$: obj
```

then, the system prompts you to enter your object‘s meta-data,

```
Enter metadata:
```

and after you have entered the meta-data, you also have to enter a name for your object:

```
Enter String:
```

The object now gets routed through the network to its destination.
In my example, this looks like that:

```
$: obj

Enter metadata: 2,2,1,3,2,4,5

Enter String: Object 1
$: Object added at /192.168.0.1 port 10020
```

Searching an object

The syntax for starting a query is almost the same as for inserting an object: First, type ‚query‘,

```
$: query
```

then, the system prompts you to enter the meta-data of the object you are looking for,

```
Enter metadata:
```

and after you have entered the meta-data, you also have to enter the name of the requested object:

```
Enter String:
```

Again, the listing of my example:

```
$: query

Enter metadata: 2,2,1,3,2,4,5

Enter String: Object 1
Requested object found by /192.168.0.1:10020
  object is owned by /192.168.0.1:10020
$:
```

Execute the test routine

To execute the test routine, type ‚test‘:

```
$: test
```

For a description of the test routine see Section 2.10.

4 Conclusions and Future Work

The goal of this semester thesis was the creation of an implementation architecture for SHARK and its transformation into a working program. The program should implement the basic mechanisms to construct the search hierarchy, to insert nodes and objects, and to route toward objects based on their meta-data description.

The implementation presented in this document meets these demands. Its functionality has been tested successfully, and even though time constraints did not allow an implementation of additional functionality, the goal of the semester thesis was fully achieved.

Distinct functionalities of the code have been tested along the development. The final implementation ran successfully in a small setting and proves the validity of the SHARK concept. After further code improvements as described below, more thorough testing on a larger scale is required to ensure correctness of the implementation.

The code has been written to prove the validity of the SHARK concept. For continued use in development or even deployment, it would benefit from the following improvements:

- Redesign of the exception handling. A consistent exception handling is the precondition for extensive testing.
- Changes of the data structures in the implementation that differ from the specification. Before new functionality is added to the code, it should be ensured, that the existing code exactly suits to the specification.
- Implementation of a convenient user interface. The one implemented here suits well for an interactive testing of simple SHARK networks, but its functionality is quite limited.

For further development of the program, the following tasks are to be carried out next:

- Adding of functionality that makes the system more dynamic: node removal, object removal, routing table repair and GoI split.
- Extensive testing of the system in bigger networks to evaluate the impact of the built-in redundancy, especially in the GoIs (as random networks) - this was not possible with the networks of the size that were used to test the algorithm up to now.
- Implementation of a real file-sharing application over SHARK, to see SHARK‘s usability in its intended environment.

5 Appendix

5.1 Deviations of the Implementation from the SHARK Specification

As described in the its description in Section 2, the implementation does not always suit exactly to the specification of SHARK [5]. The deviations are summarized in Table 1.

Table 1: Deviations of the Implementation from the SHARK Specification

Use Case / Structure	**Description of Deviation**
Node insertion	(1) Receiving of a insert_node_reply message precedes the insert_node_reply_rn message. (2) Ping-pong mechanism is not implemented.
Query	When looking for an object matching the query, the name of the object (string) is ignored.
Meta-data	(1) The positions of the values are implicitly indicated. (2) The String that definitely describes the object is not part of the meta-data structure, but handled seperately.
SHARK object	The hash value an object is calculated from its name instead of its binary data.
SHARK messages	Acknowledgment message has been added as an additional message type (type ACK, encoded as 0x99)

6 References

[1] J. Mischke, B. Stiller: *Peer-to-Peer Search with SHARK: Symmetric Redundant Hierarchy Adaption for Routing of Keywords*; ETH Zurich, Switzerland, TIK-Report Nr. 164, February 2003.

[2] J. Mischke, B. Stiller: *Design space for Disributed Search (DS)^2 - A System Designers‘ Guide*; ETH Zurich, Switzerland, TIK Report 151, September 2002.

[3] J. Mischke, B. Stiller: *Peer-to-Peer Overlay Network Management Through AGILE: An Adaptive, Group-of-Interest-based Lookup Engine* (Extended Version); ETH Zurich, Switzerland, TIK Report Nr. 149, August 2002.

[4] S. Mehr: *Lookup in Peer-to-Peer Networks;* ETH Zurich, Switzerland, Semester thesis, June 2003.

[5] J. Mischke, B. Stiller: *Specification of a Scalable Peer-to-Peer Search Infrastructure - Extended Version*; ETH Zurich, Switzerland, Draft for TIK Report 176, June 2003.

7 List of Figures

Figure 1: Software architecture overview 4
Figure 2: SHARK MSC Node Insertion [5] 6
Figure 3: SHARK MSC Object Insertion [5] 8
Figure 4: SHARK MSC Query [5] 9
Figure 5: 1- and 2-dimensional representation of the meta-data 9
Figure 6: SHARK Message Format [5] 11
Figure 7: SHARK message payload formats 11

8 List of Tables

Table 1: Deviations of the Implementation from the SHARK Specification 20